PhatKV 最适合文件系统元数据的存储引擎?

> PhatKV是一个专门为文件系统元数据场景优化的单机存储引擎，在msst 24中被提出。
>
> 本文对其进行了介绍，也加入了本人的一些理解
>
> 原论文：https://www.msstconference.org/MSST-history/2024/Papers/msst24-9.3.pdf

背景

新兴的分布式文件系统流行于采用KV模型来存储其元数据，原因主要有

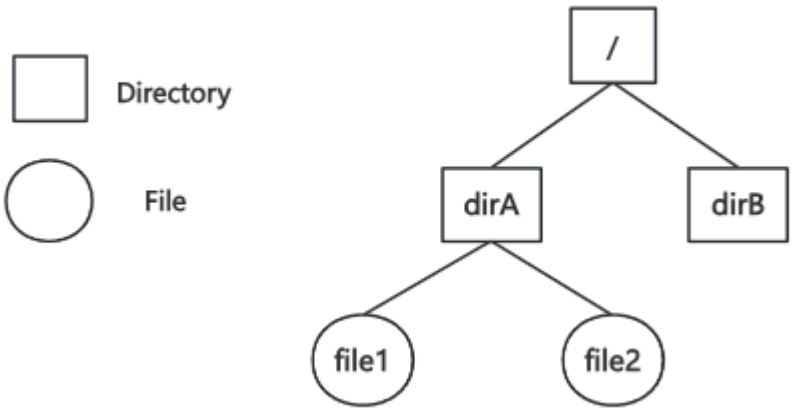
- 如今已经有很多开源且成熟的分布式KV、NewSQL系统，它们自身就具备大规模扩展的能力，
- 提供ACID的事务语义。

基于这些服务构建文件系统元数据服务可以很大程度利用它们提供的scalability，ACID事务也让支持各种元数据操作变得简单。

而为了高效地支持rename，在KV中使用*parent_inode+file_name*作为主键来构建文件系统元数据几乎已经成了最佳实践。

文件系统元数据 on KV

如下图所示的目录树结构：



添加图片注释，不超过 140 字（可选）

在KV模型中会被表示为(为了方便展示，表示为数据表结构)：

parent_inode	name	type	inode

0	dirA	directory	1
0	dirB	directory	2
1	file1	file	3
1	file2	file	4

primary_key: <parent_inode,name>

典型元数据操作

1. 要查找/dirA/file1文件是否存在，以SQL表示为（KV同理）：

SELECT inode FROM tables WHERE parent_inode=0 and name='dirA' and type='directory' #get 1

SELECT inode FROM tables WHERE parent_inode=1 and name='file1' and type='file' #get 3

于是得出该文件存在，且inode为3

2. 在/dirA中create一个文件file3，以SQL表示为：

SELECT inode FROM tables WHERE parent_inode=0 and name='dirA' and type='directory' #get 1

INSERT INTO tables (parent_inode, name, type, inode) VALUES (1, 'file3', 'file', 5)

如果INSERT成功，说明/dirA/file3创建成功。

3. 比如在/dirA进行一个ls（readdir）以SQL表示为：

SELECT inode FROM tables WHERE parent_inode=0 and name='dirA' and type='directory' #get 1

SELECT * from tables WHERE parent_inode = 1;

利用<parent_inode, name>索引的最左匹配原则，我们可以轻松得到所有parent_inode=1的数据

论文在类似这么一个kv-based的文件系统上运行了一些workload，得出了各个KV操作的比例

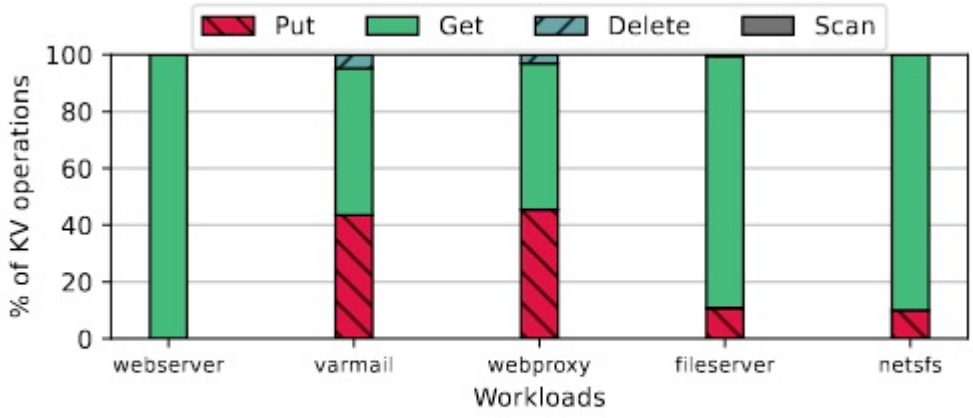


Fig. 1. **Proportion of different KV operations.** This figure illustrates the distribution of various KV operations across different file system workloads.

可以看到点查询Get操作在大多数workload下都占了大头，如果只考虑读操作的话，Get几乎占了100%，而Scan没影了。这是因为几乎所有元数据操作都需要首先通过多次Get来找到parent_inode（一般称之为Lookup），再基于这个结果来进行增删改。而Scan操作只有在readdir时才会被调用。

基于这样的一个workload，我们需要考虑一个问题就是：什么样的存储引擎才最适合文件系统元数据负载?

众所周知，单机存储引擎近乎可以分为两大类：以RocksDB为代表的LSM类，和以WiredTiger、InnoDB为代表的B族树类。特别是各种分布式KV、share-nothing NewSQL系统，在Google BigTable的先驱之下，几乎清一色选择了LSM Tree类的单机引擎。

而无论是B Tree还是LSM Tree，它们都是有序的数据结构，虽然可以支持readdir，但并不能很好地服务于Point Get，特别是LSM Tree的Merge Iterator，在各层overlap较多和墓碑多的时候更废了。

总而言之，**为了处理占比1%都不到的scan而选择有序数据结构，进而损害占比99%得Point Get的性能，这是一件很不划算的事情**。

那使用哈希引擎行不行？哈希结构肯定是能最好服务于Point Get workload的，但是纯粹的哈希结构无法服务readdir，因为一个*WHERE parent_inode = 1*操作将会导致全表扫描。

难道想要readdir就必须得是有序结构嘛？论文重新审视了readdir的posix标准，发现readdir的结果并不要求按照文件名顺序的，只需要把结果集返回就可以了：

> The ls utility shall write to standard output the names of files in the directory specified by the file operand.

也就是说，如果存在一个理想的文件系统元数据引擎，这个引擎应该：

- 高效支持 point get by <parent_inode, name>
- 高效支持 根据parent_inode找到所有的子项

基于此，论文提出PhatKV，一个专门用于文件系统元数据存储的KV引擎。

核心思想我用几句话来概括一下：

> 构建两个哈希索引
>
> 第一个哈希索引的key是*parent_inode+name*, value是entry，用于服务Point Get；
>
> 第二个哈希索引的key是parent_inode，value是第一个哈希索引，可以通过parent_inode找到所有其下的entry，用于服务ReadDir。

讲完。只想了解核心思想的话看到这里就结束了，接下来都是一些实现细节，比如怎么控制内存使用量，怎么组织磁盘结果等

设计

数据结构

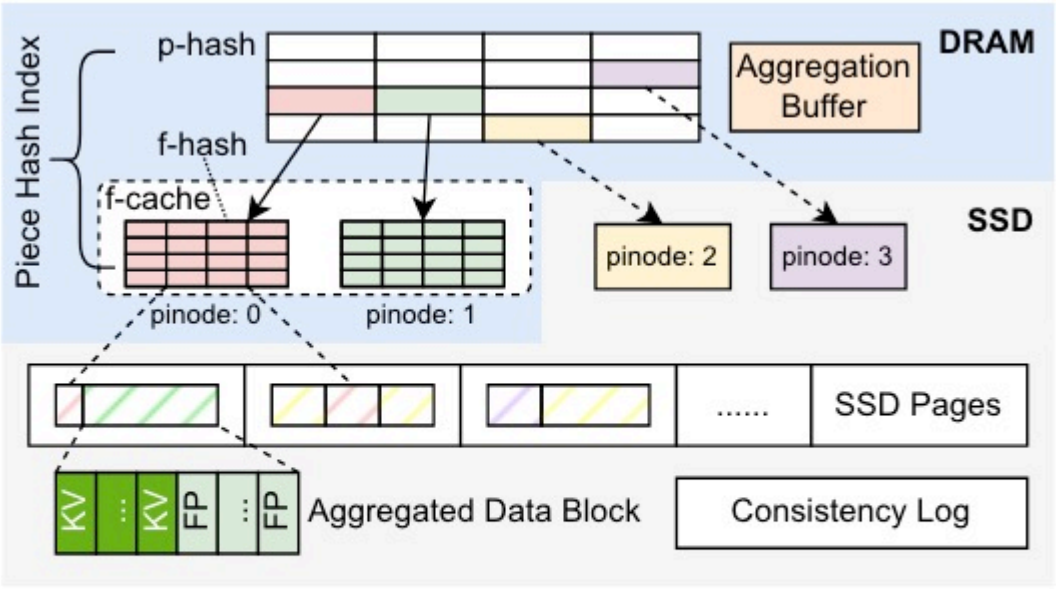


Fig. 5. **Overall structure of PhatKV.** In this figure, the piece hash index contains 4 f-hash structures. The f-hash of pinode 0 and 1 are in the f-cache, while those of pinode 2 and 3 are stored on SSD. Pinode 0 has two data blocks on SSD.

p-hash

最顶层有一个哈希索引，称为Pinode Hash（p-hash），key为parent_inode, value指向f-cache。这个哈希索引处于内存中，使用CLHT（Cache-Line Hash Table）作为实现，原因是：

1）目录操作（例如 statdir、mkdir 和 rmdir）在实际文件系统工作负载中并不常见，因此 p-hash 不太可能被修改;

2）目录的数量通常小于文件的数量，因此p-hash的大小相对较小。论文实验结果表明，800 万个目录的 p-hash 仅消耗 256 MB 内存

f-hash

f-hash（FileName Hash）也是一个哈希索引，key为filename，value为指向硬盘上 or 内存上 一段数据的指针（PageID+Offset）。每个f-hash都存储了对应的parent_inode下所有的子项索引，系统中会有若干个f-

hash。

f-hash是混合介质存储的，可以位于硬盘，也可以位于内存中，内存中管理f-hash的模块叫f-cache，通过LRU来做缓存淘汰。

一个f-hash的结构如图所示，内存和硬盘上的结构一致，所以没有序列化开销。

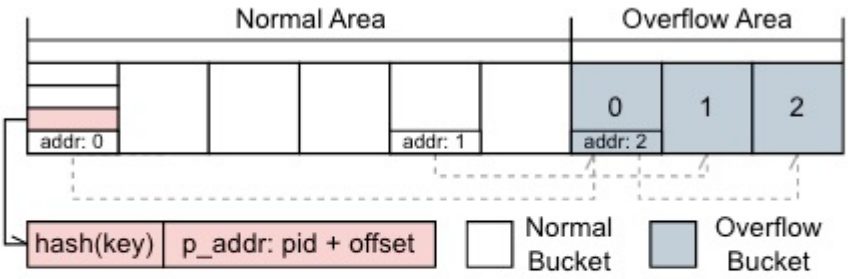


Fig. 6. **Structure of f-hash.** In this figure, the f-hash contains 6 normal buckets and 3 overflow buckets. Bucket 0 links two overflow buckets, and bucket 4 links 1 overflow bucket.

f-hash的每一项大小为16字节：

- 8字节：hash（name）
- 8字节：page_id+offset

f-hash的normal area有若干个bucket，每个bucket可以存储4个项（64字节，Cache Line对齐），图中的这个f-cache有6个normal bucket，一共占用64*6=384字节。

overflow area是用来解决哈希冲突的，图中Bucket 0 链接2个overflow bucket，Bucket 4 链接1个overflow bucket

当f-hash从f-cache中被淘汰时，它需要被写入ssd，但如果f-hash的大小不足一个ssd page，那么会与其他f-hash merge起来一起下刷。同样的，如果f-cache的大小超过一个ssd page，则会被分块，以块为单位来做evict。

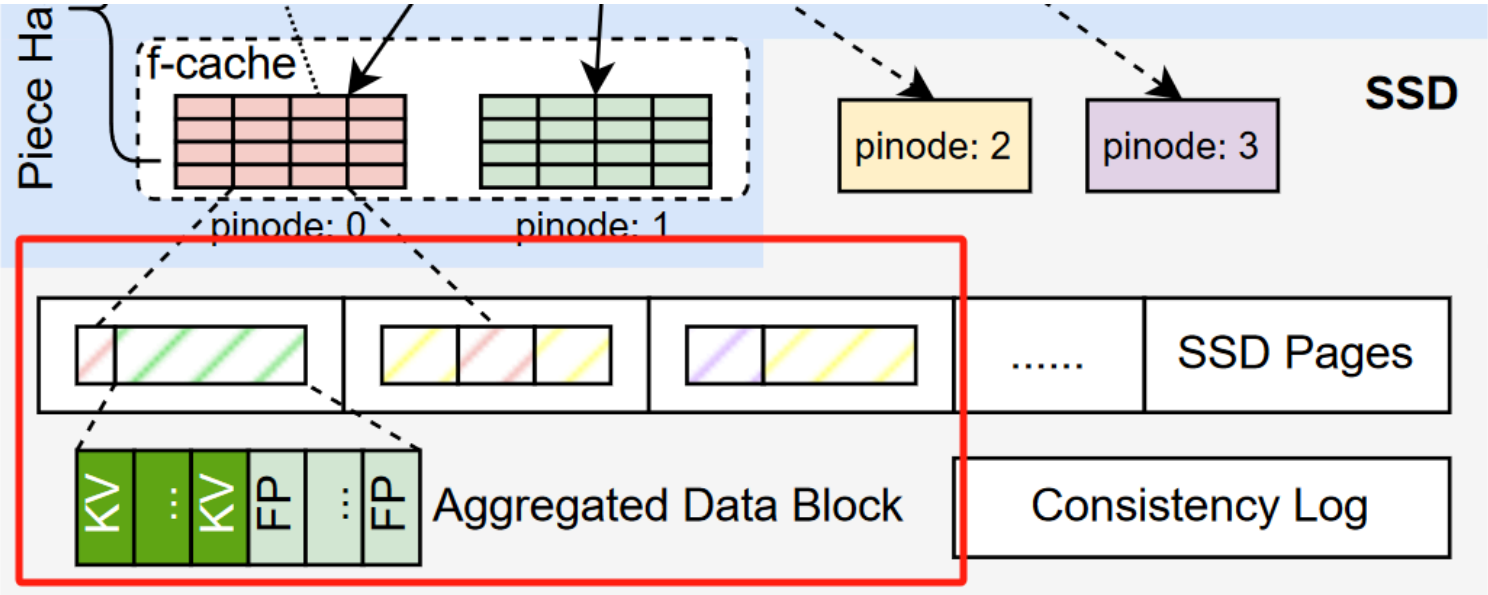
on-disk

f-hash中value指向的元数据称为一个entry，同一个parent_inode下的若干entry组成为一个数据块，一个数据块必然只包含一个parent_inode下的子项，一个parent_inode下可以包含若干个数据块。

每个数据块：

- 必然存着同一个pid下的数据
- <= 4kb, 不定长

一个数据块的存储格式如图所示

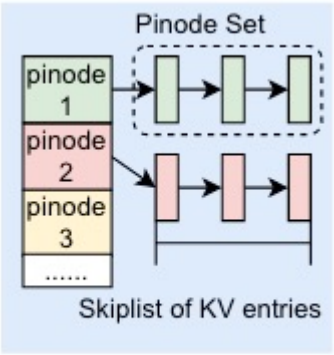


fp 数据包含数据块内每个 KV 项的 2 字节指纹值，用于垃圾回收时快速判断两个key是否相同。数据块的最大大小限制为一个SSD Page

聚合

内存聚合 Memory Aggregation

PhatKV有类似memtable的概念，新写入的数据会先保留在memtable中，memtable按照parent_inode进行分组，如图 所示。



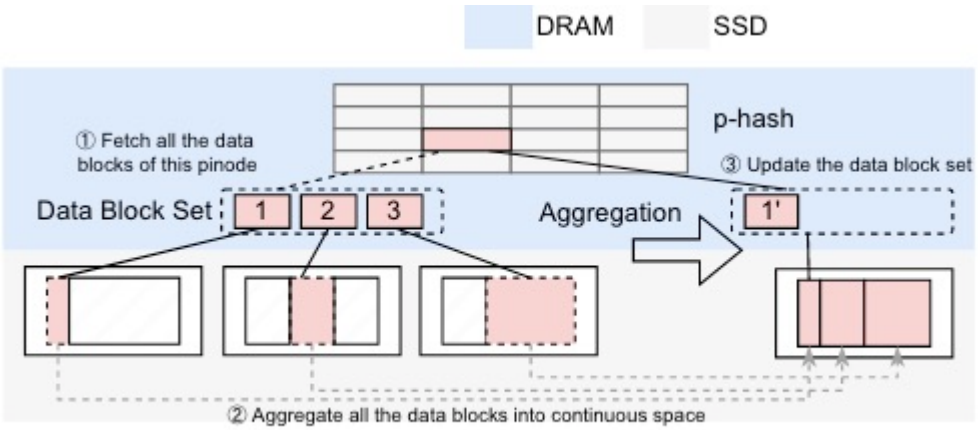
(a) Memory Aggregation

对于每个 parent_inode，都会对其数据用skiplist维护起来。

一旦整个memtable中KV项的总大小（64MB）或某个pinode集的大小超过阈值（4KB），就会触发后台聚合线程。聚合线程从memtable中选择大小最大的pinode集合，构造数据块，并将这些数据块写入SSD。每当一个新的数据块被持久化时，这些KV项的相关KV索引条目就会被更新到相应的f-hash中。

磁盘聚合 Disk Aggregation

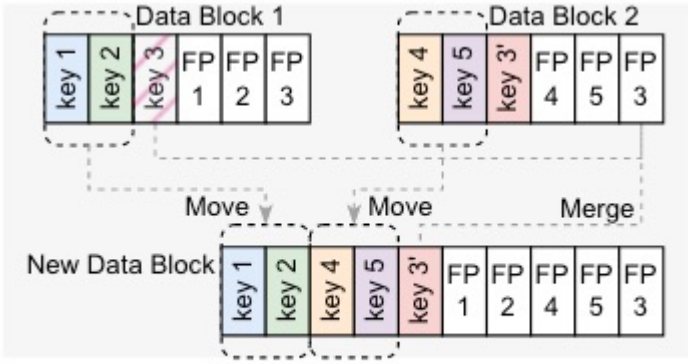
一个parent_inode下的数据块集可能在磁盘上离散分布，磁盘聚合就是将一个parent_inode下的离散的数据块集merge成一个连续的，再rewrite下去



(b) Disk Aggregation

深度聚合 Deep Aggregation

其实就是GC，因为每个数据块也是immutable的，所以随着数据的更新，旧数据就会成为墓碑，当墓碑比例过大，触发一次GC，可以根据指纹值fp快速判断key是否相同



(c) Deep Aggregation

读写流程

create file：

1. 根据pid从p-hash中找到对应的f-hash。
2. 写WAL
3. KV写入memtale中，然后写入f-hash中

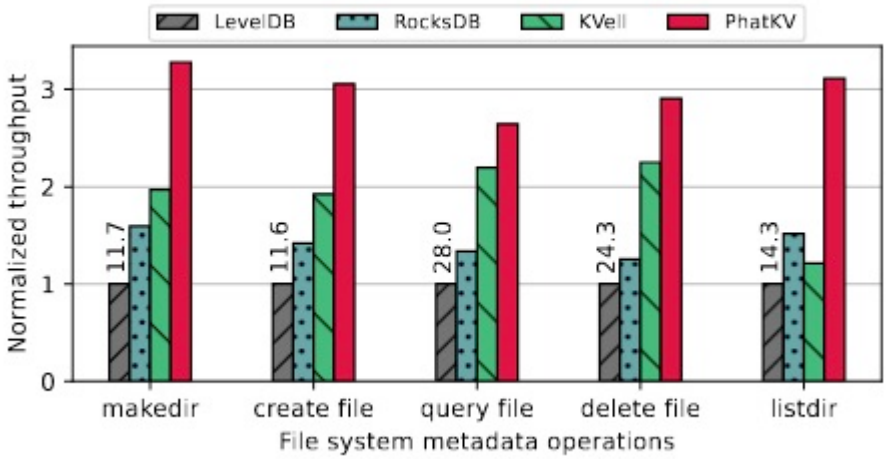
get：

1. 根据pid从p-hash中找到对应的f-hash
2. 从f-hash中，根据filename找到kv entry的位置（在磁盘or在memtable）

其他

1. 用WAL来保证一致性。(但没说怎么snapshot然后truncate log
2. io_uring

EVALUATION



没有提的点:

1. f-hash的扩容?
2. Disk aggregation后批量改f-hash，反压前台写入?
3. 怎么高效snapshot?

discuss

论文内容上:

1. f-hash的扩容?
2. Disk aggregation后批量改f-hash，反压前台写入?

功能+性能:

1. 业务是否真的不需要有序？会不会有用户在本地文件系统上养成惯性了?
2. 引擎latency占全流程的多少？需要有个breakdown来预估收益

集成到tafdb上的问题:

1. 如何高效和低成本地支持snapshot和iterator

2. 如果按照现在一个store多个tablet模型的话，这个引擎需要支持